Accelerating Sparse Matrix-Vector Multiplication on GPUs using Bit-Representation-Optimized Schemes

Wai Teng Tang*, Wen Jun Tan*, Rajarshi Ray†, Yi Wen Wong†, Weiguang Chen†,
Shyh-hao Kuo‡, Rick Siow Mong Goh‡, Stephen John Turner*, Weng-Fai Wong†
*School of Computer Engineering, Nanyang Technological University, Singapore
†Department of Computer Science, School of Computing, National University of Singapore, Singapore
‡Institute of High Performance Computing, Agency for Science, Technology and Research, Singapore

ABSTRACT

The sparse matrix-vector (SpMV) multiplication routine is an important building block used in many iterative algorithms for solving scientific and engineering problems. One of the main challenges of SpMV is its memory-boundedness. Although compression has been proposed previously to improve SpMV performance on CPUs, its use has not been demonstrated on the GPU because of the serial nature of many compression and decompression schemes. In this paper, we introduce a family of bit-representation-optimized (BRO) compression schemes for representing sparse matrices on GPUs. The proposed schemes, BRO-ELL, BRO-COO, and BRO-HYB, perform compression on index data and help to speed up SpMV on GPUs through reduction of memory traffic. Furthermore, we formulate a BRO-aware matrix reordering scheme as a data clustering problem and use it to increase compression ratios. With the proposed schemes, experiments show that average speedups of 1.5× compared to ELLPACK and HYB can be achieved for SpMV on GPUs.

Categories and Subject Descriptors

E.4 [**Data Storage Representations**]: Data compression; D.1.3 [**Programming Techniques**]: Parallel programming; C.1.2 [**Processor Architectures**]: Graphics Processing Units (GPU)

General Terms

Algorithms, Compression, Experimentation, Performance

Keywords

Sparse matrix format, data compression, matrix-vector multiplication, GPU, parallelism, memory bandwidth.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC '13, November 17 - 21 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11...$15.00.
http://dx.doi.org/10.1145/2503210.2503234

1. INTRODUCTION

Sparse systems arise naturally in many engineering problems, such as image processing, circuit analysis and structural mechanics. These systems are normally represented by sparse matrices with efficient storage schemes that store only the non-zero elements of the matrices, and are usually solved using iterative algorithms such as the Conjugate Gradient (CG) and Generalized Minimum Residual (GMRES) methods [21]. An important component, and also the main bottleneck of these iterative algorithms is the sparse matrix-vector (SpMV) multiplication kernel, which is used repeatedly until the iterative process reaches convergence.

Different optimization strategies and data formats such as Compressed Sparse Row (CSR) and ELLPACK-ITPACK (ELLPACK) have been proposed by researchers which are suitable for different kinds of matrix structure and different hardware architectures [27, 15, 8]. For instance, the ELLPACK format is suitable for computing the SpMV product on vector processors [15]. Because SpMV is inherently a memory bound operation, lossless compression methods have recently been proposed to reduce the memory bandwidth usage of SpMV computations on the CPU. Willcock and Lumsdaine [26] developed compression schemes based on hybrid delta and run-length encoding for the Compressed Sparse Row (CSR) format, and reported performance improvements of 30%. Likewise, Kourtis et al. [16] employed index and value compression to reduce memory traffic and achieved speedups of close to 1.2×.

With the emergence of GPUs as a popular platform for scientific computing over the past few years, the SpMV operation can now be offloaded to GPUs to exploit the massive parallelism offered by their many-core architecture. As such, much work has gone into optimizing the SpMV operation for the GPU. Baskaran and Bordawekar [2] investigated efficient thread mapping and data access strategies for SpMV kernels on GPUs. Bell and Garland [4] explored several efficient implementation techniques for SpMV multiplication using the Compute Unified Device Architecture (CUDA) [19] programming model. They designed an efficient hybrid storage format that combines the benefits of the coordinate and ELLPACK formats with good performance over a large collection of sparse matrices. More recently, newer strategies such as Sliced-ELLPACK [18] and ELLPACK-R [23] have been proposed to improve the performance of SpMV computation on GPUs.

However, all of these formats do not employ any lossless compression schemes to reduce memory traffic. Unlike in the

case of the CPU, compression schemes such as run-length encoding are difficult to implement on the GPU due to its warp-based execution model. In addition, many entropy compression schemes are serial in nature and are difficult to implement efficiently on GPUs because each data element depends on the preceding data. This dependency makes parallelization on the *single instruction multiple thread* (SIMT) architecture of the GPU hard. Nevertheless, in this paper, we will demonstrate that improved SpMV performance can be achieved with compression schemes that are specially tailored for the GPU. Our proposed bit-representation-optimized (BRO) compression schemes, which include BRO-ELL, BRO-COO and BRO-HYB, reduce storage size by reducing the number of bits required to represent a sparse matrix.

Our contributions in this paper are: (i) we designed effective and GPU-optimized schemes to further compress existing sparse matrices formats; these schemes can also be used to improve the performance of SpMV on GPUs, (ii) we developed a matrix reordering method that can be used to increase the compression ratio, and finally, (iii) we evaluated the effectiveness of our approach using a large set of sparse matrices and showed that performance speedups of between 1.2× and 2.1× relative to ELLPACK and HYB can be achieved.

The remainder of the paper is structured as follows. In Section 2, we present background material on commonly used storage formats for computing the SpMV product on GPUs. Section 3 presents our proposed BRO schemes for computing the SpMV product. Experiment results are presented in Section 4, followed by a discussion of our results. Section 5 discusses prior research which is related to ours, and Section 6 concludes the paper.

2. BACKGROUND

Sparse matrix-vector multiplication on GPUs requires implementations that are carefully optimized for the underlying graphics hardware, of which the architecture is massively threaded and significantly different from general CPU architectures. For example, for the Nvidia Fermi GPU architecture, each executable GPU kernel is launched with a fixed number of threads organized into a grid of thread blocks. The threads in each thread block are executed synchronously in units known as *warps*, with each warp containing a group of 32 threads. The warps of each thread block are scheduled and executed on *streaming processors* on the GPU device.

The GPU also contains off-chip dynamic global memory with high bandwidths. However, a high memory throughput is attainable if the global memory is accessed in a coalesced manner, i.e. threads in a warp should access contiguous locations of the memory. Achieving a good performance on GPU requires latencies due to memory transfers to be properly hidden by arithmetic computation. Another important optimization on the GPU is to avoid control flow divergence arising from branch statements in the kernels. Since threads in a warp are executed simultaneously, a branch divergence would cause statements on both branches to be serially executed, resulting in computational inefficiencies.

Optimized and efficient implementations for computing the SpMV product on GPUs have been developed in prior works. Such optimized implementations often depend on the way the sparse matrix data is stored and layout in memory, and may also include additional information to optimize computation. They include various schemes such as the ELLPACK, hybrid (HYB), and ELLPACK-R formats. In the next section, we describe some of these storage formats in greater detail.

2.1 Sparse Matrix Storage Formats

2.1.1 Coordinate format (COO)

The classical COO coordinate format stores both the row and column indices of the non-zero elements explicitly in two separate arrays. For a $m \times n$ matrix with nnz non-zeros, the row and column indices take up $2nnz$ of storage space. For example, consider the following matrix,

$$A = \begin{pmatrix} 3 & 0 & 2 & 0 & 0 \\ 2 & 6 & 5 & 4 & 1 \\ 0 & 1 & 9 & 0 & 7 \\ 0 & 0 & 0 & 8 & 3 \end{pmatrix},$$

the COO format stores the sparse data in three arrays - *row_idx* stores the row index of each element, *col_idx* stores the column index of each element and *vals* stores the non-zero elements.

$$\begin{aligned} row\_idx &= [1,1,2,2,2,2,2,3,3,3,4,4] \\ col\_idx &= [1,3,1,2,3,4,5,2,3,5,4,5] \\ vals &= [3,2,2,6,5,4,1,1,9,7,8,3] \end{aligned}$$

The implementation in [5] divides the data into intervals, and a single warp will access the corresponding data in the arrays to compute the matrix-vector product for each interval. A parallel segmented reduction operation is then performed by each of the warps to add up products that reside on the same rows.

2.1.2 ELLPACK-ITPACK format (ELLPACK)

The ELLPACK format is well suited to SIMD and vector architectures. It stores the non-zero elements in a dense 2D array by shifting and packing them towards the left as shown below. A total of $2m \times k$ storage space is required, where k is the maximum number of non-zeros in a row. In sparse matrices, storage is conserved because k is usually much smaller than n.

$$col\_idx = \begin{pmatrix} 1 & 3 & * & * & * \\ 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 5 & * & * \\ 4 & 5 & * & * & * \end{pmatrix}, vals = \begin{pmatrix} 3 & 2 & * & * & * \\ 2 & 6 & 5 & 4 & 1 \\ 1 & 9 & 7 & * & * \\ 8 & 3 & * & * & * \end{pmatrix}$$

For the same matrix A as above, two dense 2D arrays are used in ELLPACK - *col_idx* stores the column index of each non-zero element, and *vals* stores the non-zero values. Entries in the arrays that do not contain valid data are marked using '*'. The implementation in [5] stores the 2D matrices using column-major order and maps a thread to each row so that the column index and values data can be accessed in a coalesced manner.

2.1.3 Hybrid format (HYB)

Because of the additional padding required to pad each row to length k, the maximum number of non-zeros in a row, the ELLPACK format is not efficient when the number of non-zeros in each row varies substantially from the average number of non-zeros per row. When this happens, there is wastage in terms of the total storage required, as well as

computational resources allocated to the padding elements marked '*'. To deal with such matrices, Bell and Garland [4] introduced the hybrid storage format, HYB, that partitions a matrix into a ELLPACK component and a COO component. For instance, the matrix in the above example may be partitioned in the following way.

$$\text{ELLPACK} \begin{cases} col\_idx = \begin{pmatrix} 1 & 3 & * \\ 1 & 2 & 3 \\ 2 & 3 & 5 \\ 4 & 5 & * \end{pmatrix}, vals = \begin{pmatrix} 3 & 2 & * \\ 2 & 6 & 5 \\ 1 & 9 & 7 \\ 8 & 3 & * \end{pmatrix} \\ \text{COO} \begin{cases} row\_idx = [2, 2] \\ col\_idx = [4, 5] \\ vals = [4, 1] \end{cases} \end{cases}$$

In the implementation in [5], a large sparse matrix is partitioned using a heuristic, where the dividing column k is determined such that the number of rows with at least k non-zeros is less than a third of the total matrix rows.

2.1.4 ELLPACK-R format

The ELLPACK-R format improves upon ELLPACK by introducing an additional array that stores the number of non-zeros in each matrix row. For matrix A, apart from the *col_idx* and *vals* arrays which are the same as ELLPACK, ELLPACK-R also records the length of each row of *col_idx* or *vals* in the *row_length* array,

$$row\_length = [2, 5, 3, 2].$$

This additional information incurred by ELLPACK-R results in several benefits over ELLPACK. Since the length of each row is known, the inner loop which performs the multiply-add operation does not need to include conditional branches to check for padded elements. Moreover, useless iterations and computation on the padded elements can be reduced by each thread, especially when *row_length*[i] for each ith row is small compared to k, the largest non-zero row length. Thus, it is not necessary for every thread to loop through the full k number of iterations, and the time required by each thread is only limited by the longest computing thread within the same warp.

3. BIT REPRESENTATION OPTIMIZATIONS

Sparse matrix-vector multiplication is a memory-bound operation that has low arithmetic intensity. Consider classical storage formats such as COO and ELLPACK that are used to represent sparse matrices. Data structures such as these typically employ index arrays for indirect access to the dense input vectors. For every multiply-add operation, ELLPACK requires three memory accesses, whereas for COO, four accesses are needed. Because of the low ratio of floating point operations to the number of memory accesses, performance of the SpMV kernel is usually limited by the memory bandwidth available on the GPU.

Therefore, we propose using a family of efficient compression schemes (BRO-ELL, BRO-COO and BRO-HYB) to reduce memory traffic within the GPU. The main idea behind these schemes is to compress the index data, which usually contains a large amount of redundant information, using an efficient bit representation format. Reducing the number of bits required to store the index data may then improve the performance of SpMV on GPUs by reducing its memory bandwidth usage. Because of the fact that the sparse matrix is usually fixed and does not change across iterations in many of the iterative algorithms, the compression can be done offline on the host CPU, whereas decompression has to be performed online on the GPU, before computation of the matrix-vector product.

In the design of the BRO storage schemes, we took into consideration two important aspects. The first is that the decompressor residing on the GPU must be relatively lightweight in comparison to the multiply-add operations of SpMV so that many of the precious GPU cycles are allocated for useful work, and not used up solely to decompress the index data. Second, unlike on the CPU which contains complex hardware for branch prediction, the compressed formats should be relatively easy to decompress on the GPU without incurring costly warp divergence penalties. As such, prior schemes that were used for CPUs [26, 16] cannot be directly applied on GPUs. Likewise, schemes such as run-length coding are unable to meet our objective of lightweight decoding without branch divergence.

3.1 The BRO-ELL Scheme

The schematic diagram in Fig. 1 provides a summary of the operations performed in the BRO-ELL scheme, as well as the steps required to obtain a compressed stream for use later during SpMV computation. Figure 1 further illustrates each step that is applied using the example matrix A in Section 2.

As shown in Fig. 1, the main idea behind the scheme is to compress the 2D column index array (*col_idx*) using bit packing. Let $[c_{i,j}]$ represent elements of the n-by-k array *col_idx*. There are several stages to the compression scheme. First, the column index values are preprocessed into another array, $del\_idx = [\delta_{i,j}]$, using the following delta encoding scheme, $\delta_{i,j} \leftarrow c_{i,j} - c_{i,j-1}$, with the $c_{i,-1}$ values initialized to zero. Delta coding is a preprocessing stage that is commonly used in many signal compression schemes. In our case, it also serves to reduce redundant information present in the index data and makes it more compressible in later stages. Note that the delta values will be positive since the column index values are monotonically increasing. The zero value is used to denote invalid data.

Next, *del_idx* is partitioned into slices with equal height, h, and each slice is compressed independently of other slices. Each of the slices can be assigned to a thread block, in which case h represents the number of threads in a thread block. In our current implementation, we used a thread block size of 256. For every slice, since the number of columns containing valid data may be different, an array *num_col* is introduced to record the actual length of each slice, i.e. $num\_col = [l_1, l_2, ..., l_s]$, where s is the total number of slices. By using this array, extraneous computations can be reduced especially when the length of different slices vary substantially from the full column length k.

Subsequently, data in each slice is then packed according to the number of bits required for each index. As illustrated in the diagram, in order to match the synchronous SIMT execution model of the GPU and ensure that data access is coalesced, a fixed number of bits is allocated for each column in a slice to store all the delta values of that column. This requires finding the maximum number of bits required to represent the values in each column. For in-

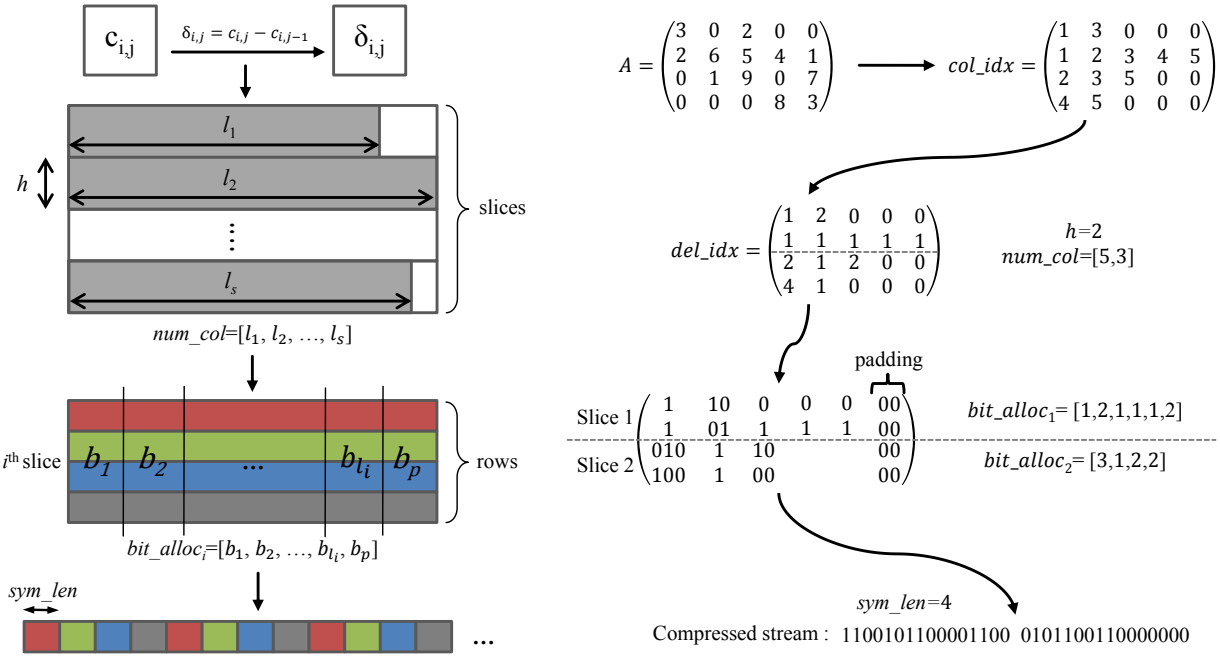
$c_{i,j}$
$\delta_{i,j} = c_{i,j} - c_{i,j-1}$
$\delta_{i,j}$
l_1
l_2
l_s
h
slices
$num\_col=[l_1, l_2, \ldots, l_s]$
i^{th} slice
b_1 b_2 ... b_{l_i} b_p
rows
$bit\_alloc_i=[b_1, b_2, \ldots, b_{l_i}, b_p]$
sym_len
...
$A = \begin{pmatrix} 3 & 0 & 2 & 0 & 0 \\ 2 & 6 & 5 & 4 & 1 \\ 0 & 1 & 9 & 0 & 7 \\ 0 & 0 & 0 & 8 & 3 \end{pmatrix}$
$col\_idx = \begin{pmatrix} 1 & 3 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 5 & 0 & 0 \\ 4 & 5 & 0 & 0 & 0 \end{pmatrix}$
$del\_idx = \begin{pmatrix} 1 & 2 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 1 & 2 & 0 & 0 \\ 4 & 1 & 0 & 0 & 0 \end{pmatrix}$
h=2
num_col=[5,3]
padding
Slice 1
Slice 2
$\begin{pmatrix} 1 & 10 & 0 & 0 & 0 & 00 \\ 1 & 01 & 1 & 1 & 1 & 00 \\ 010 & 1 & 10 & & & 00 \\ 100 & 1 & 00 & & & 00 \end{pmatrix}$
bit_alloc_1= [1,2,1,1,1,2]
bit_alloc_2= [3,1,2,2]
sym_len=4
Compressed stream : 1100101100001100 0101100110000000

Figure 1: Compression scheme for BRO-ELL.

stance, column j uses b_j bits to pack each element in that column. Therefore, an additional array for each ith slice, $bit\_alloc_i = [b_1, b_2, ..., b_{l_i}, b_p]$, is introduced to store the bit allocation information for each column, where l_i is the number of columns in the slice. Note that for each slice, the data stream in each row (row stream) will contain the same number of bits. b_p are additional bits used for padding such that sym_len divides $\sum_{j=1}^{l_i} b_j$.

In the final step, the row streams are multiplexed together with a symbol length of sym_len, usually 32 or 64 bits, which is the granularity of data access by each thread during the decompression phase. Multiplexing, depicted with colored boxes in the diagram, ensures that each GPU thread can be mapped to a row stream and that threads can access the data streams in a coalesced manner. All the above steps are performed offline on the host CPU. Figure 1 demonstrates how matrix A is compressed and packed, assuming $h = 2$ and $sym\_len = 4$. In this example, the final size of the compressed stream is 32 bits. Compare this with the original col_idx array which required 80 bytes of storage.

On the GPU, the decompressor has to decode the bits prior to performing the multiply-add operation. Algorithm 1 shows the pseudocode for decompressing the compressed data stream on-the-fly and computing the SpMV on the GPU. In the current implementation, each slice is mapped to a thread block and scheduled on a streaming multiprocessor. A single thread is mapped to each row in the slice. As with other formats [2, 4, 10, 18], the texture cache is used to cache access to input vector x.

To understand the decompression process conceptually, threads in each of the slices can be viewed as synchronously performing a load-and-decode-symbol cycle, followed by computing a multiply-add operation with the decoded index. The main loop will iterate as many times as there are columns in a slice. In each iteration, the number of bits required to decode the next symbol is read into b. If b is less than the number of remaining bits in the symbol buffer (rb), then there is no need to read in the next symbol. Otherwise, the next symbol has to be loaded from the compressed stream. Recovering the column index is a simple matter of extracting b bits from the symbol buffer and accumulating the decoded index. In the current implementation, the bit allocation arrays can be allocated in the constant memory since every thread within a block will access the same data during each iteration.

Unlike CPU methods which require multi-way branching to decode the compressed symbols [26, 16], the advantage of this scheme lies in the elimination of costly divergent branches that become serialized during warp execution, and instead replacing them with an efficient load-decode-and-compute process. In each iteration, all the threads in a warp will either take the first branch or the second branch of the conditional statement, thereby ensuring identical control flow and avoiding warp divergence. Reduction in memory traffic is also achieved since a large number of iterations take the first branch. The second branch where data is loaded from global memory is only taken when the symbol buffer is depleted. Furthermore, coalesced access of the compressed data is achieved through multiplexing of the compressed bit streams in each row.

3.2 The BRO-COO Scheme

Our proposed BRO-COO scheme employs a similar compression method as BRO-ELL. In the COO storage format, there are two index arrays, one for the row indices and the other for the column indices. BRO-COO seeks to reduce information redundancy only in the row index array row_idx. Unlike BRO-ELL, BRO-COO uses the notion of intervals and not slices because the data for COO is organized using 1D arrays and not 2D arrays. As illustrated in Fig. 2, the

Algorithm 1 - BRO-ELL decompression and SpMV multiplication routine for calculating $y = A \cdot x$.

Input: $num\_col = [l_1, l_2, ..., l_s]$ contains the number of columns in each slice, $bit\_alloc_i = [b_1, b_2, ..., b_{l_i}, b_p]$ stores the bit allocation information and $comp\_str_i$ contains the compressed stream for slice i, *vals* contains the matrix values, and x is the dense input vector.
Output: Dense vector y containing the result of $A \cdot x$.
Definition: Let *sym* denote a bit buffer with a size of *sym_len* and $sym[p : p+q]$ refer to q bits from p to $p+q-1$ in the buffer. If q is zero or negative, then zero is returned. Let $comp\_str_i[m]$ refer to the mth symbol in the compressed stream.
Method:

1: tid = thread ID, h = block size, i = block ID
2: $row\_idx = i \times h + tid$
3: $col\_idx = rb = sym = decoded = sum = 0$
4: **for** $c = 0$ **to** $l_i - 1$ **do**
5: $b = bit\_alloc_i[c]$
6: **if** $b < rb$ **then**
7: $decoded = sym[0 : b]$
8: $rb = rb - b$
9: **else**
10: $decoded = sym[0 : rb]$
11: $b = b - rb$
12: $sym = comp\_str_i[c \times h + tid]$
13: $decoded = decoded \ll b + sym[0 : b]$
14: $rb = sym\_len - b$
15: **end if**
16: $sym = sym \ll b$
17: **if** $decoded \neq invalid$ **then**
18: $col\_idx = col\_idx + decoded$
19: $sum = sum + vals[row\_idx, col\_idx] \times x[col\_idx]$
20: **end if**
21: **end for**
22: $y[row\_idx] = sum$

row index data is first divided into intervals. Each interval is then organized into a 2D array such that the row index increases monotonically in the vertical direction. Each interval will be processed and computed by a warp later on. As in BRO-ELL, the 2D row index array $[r_{i,j}]$ is preprocessed using delta encoding, followed by bit packing. The same number of bits is used to pack all values within the interval and the bit count is stored in the *bit_alloc* array. In the final step, the compressed row streams are then multiplexed together as in BRO-ELL. In BRO-COO, the decompression scheme is similar to that of BRO-ELL, except that each interval is processed by a warp, whereas for BRO-ELL, each slice is mapped to a thread block.

3.3 The BRO-HYB Scheme

When the number of non-zeros per row of a sparse matrix varies substantially, the matrix can be partitioned into a BRO-ELL component and a BRO-COO component. Similar to the hybrid ELLPACK-COO (HYB) format, BRO-HYB combines the BRO-ELL and BRO-COO formats by dividing a sparse matrix into BRO-ELL and BRO-COO partitions with the same algorithm as in [4, 5]. This allows evaluation and comparison of the classical and BRO formats later on in the next section.

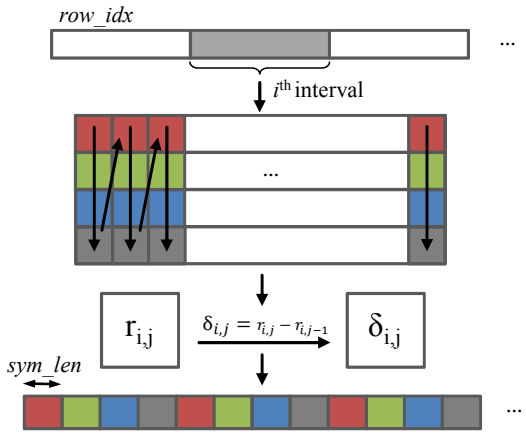

Figure 2: Compression scheme for BRO-COO.

3.4 Matrix Reordering

Several matrix reordering methods have been proposed in the past to improve the performance of sparse matrix-vector multiplication, such as those by Pinar [20] and Choi [6], while other methods such as the Reverse Cuthill-Mckee (RCM) [9] and approximate minimum degree (AMD) [1] algorithms have been developed to reduce the number of fill-ins during matrix factorization. However, these methods do not take into consideration the amount of space savings that can be achieved through reordering. In this section, we propose a BRO-aware reordering method (BAR) such that the resulting index array can be better compressed using our bit-representation-optimized format. We will then evaluate and compare the proposed method with existing techniques in the next section.

For the BRO-ELL format, the overall compression that is achievable depends on the bit allocation arrays, bit_alloc_i, which in turn depends on the range of the delta values in each column of a slice. Therefore, reordering of the rows can be employed to bring those which have similar bit allocation patterns together in order to reduce the overall storage size, and consequently reduce the number of memory transactions. Given a matrix A and the matrix-vector product $y = A \cdot x$, we define a row permutation matrix P and transform the product to $y' = A' \cdot x$, where $y' = P \cdot y$ and $A' = P \cdot A$.

The determination of P can be formulated as a data clustering problem as follows. Without loss of generality, let us assume that $h|m$, i.e. h divides m, and $v = m/h$. Also, let w be the warp size and let w divide h. Let $\mathcal{R} = \{\mathbf{r_1}, \mathbf{r_2}, ..., \mathbf{r_m}\}$ be the rows of the delta-encoded matrix *del_idx*, and $\mathbf{r_1}$,..., $\mathbf{r_m} \in \mathbb{R}^k$. The clustering problem involves finding v disjoint equi-partitions $\{\mathcal{S}_t\}_{t=1}^{v}$ of the set $\mathcal{R}$ which minimizes following objective function,

$$\Phi(\{\mathcal{S}_t\}_{t=1}^{v}) = \sum_{i=1}^{v} h/w \times \left(\lceil \frac{\sum_{j=1}^{k} d(\mathcal{S}_i, j)}{\alpha} \rceil + \sum_{j=1}^{k} c(\mathcal{S}_i, j) \right). \quad (1)$$

Overall, Eqn. (1) seeks to find a partitioning that minimizes the number of memory transactions required for the SpMV operation. The first term within the parentheses in Eqn. (1) denotes the number of memory transactions required to ac-

Algorithm 2 - Matrix reordering routine.

Input: Sparse matrix A, in terms of associated arrays *col_idx* and *vals*.
Preprocessing: Obtain *del_idx* from *col_idx*.
Output: A set of partitions $\{\mathcal{S}_t\}_{t=1}^{v}$. Each cluster $\mathcal{S}_t$ contains a sequence of rows in the order of insertion.
Method:

1: initialize $\mathcal{S}_t \leftarrow \varnothing$ for $t = 1$ to v
2: $\mathcal{D} \leftarrow$ sort rows in *del_idx* according to row length and let $\{\mathbf{r_i}\}$ be the sequence of rows in $\mathcal{D}$
3: **for** $t = 1$ **to** v **do**
4: $\quad \mathcal{S}_t \leftarrow \mathcal{S}_t \cup \mathbf{r_{(t-1)\times h+1}}$
5: $\quad \mathcal{D} \leftarrow \mathcal{D} - \mathbf{r_{(t-1)\times h+1}}$
6: **end for**
7: **for** $\mathbf{r_i} \in \mathcal{D}$ **do**
8: $\quad$ **for** $m = 1$ **to** v **do**
9: $\quad\quad$ evaluate $\Phi_m \leftarrow \Phi(\{\mathcal{S}_t\}_{t=1,t\neq m}^{v} \cup (\mathcal{S}_m \cup \mathbf{r_i}))$
10: $\quad$ **end for**
11: $\quad$ find partition p with the least cost in $\{\Phi_m\}_{m=1}^{v}$ and satisfying $|\mathcal{S}_p| < h$
12: $\quad \mathcal{S}_p \leftarrow \mathcal{S}_p \cup \mathbf{r_i}$
13: **end for**

cess the delta indices in the *del_idx* array given a symbol length of α, while the second term denotes the number of memory transactions required for accessing the x vector.

The function $d(\mathcal{S}_i, j)$ returns the maximum number of bits required to represent the jth element of all the row vectors in the partition $\mathcal{S}_i$. That is, denoting a row $\mathbf{r_a} = [r_{a1}, r_{a2}, ..., r_{an}]$ in partition $\mathcal{S}_i$, and letting $\Gamma(u)$ be a function that returns the number of bits require to pack an unsigned integer u, then

$$d(\mathcal{S}_i, j) = \max\{\Gamma(r_{aj}),\ \mathbf{r_a} \in \mathcal{S}_i\}. \quad (2)$$

The function $c(\mathcal{S}_i, j)$ returns the number of unique cachelines used for accessing the x vector by the jth element of all rows in partition $\mathcal{S}_i$. Thus, if Ω denotes a mapping from r_{aj} to the corresponding cacheline for addressing x, then

$$c(\mathcal{S}_i, j) = |\cap\, \Omega(r_{aj}),\ \mathbf{r_a} \in \mathcal{S}_i\}|. \quad (3)$$

A limitation of the above function is that it takes into account spatial locality but not temporal locality.

In general, finding the global optimum solution for data clustering is known to be NP-hard [14]. In Algorithm 2, we employ a greedy heuristic to partition the rows of a matrix according to Eqn. 1. The purpose of the sorting step (line 2) is to seed each cluster with an entry that is sufficiently separated in terms of row length. Each partition is then populated with an entry that is spaced equally apart from adjacent partitions. For each of the remaining rows in set $\mathcal{D}$, the cost of placing that row in each of the partitions is determined. The row is then placed in the partition with the least cost (viz. Eqn. 1) subject to the equi-partition constraint. If a cluster is full, then the row is placed in the next available cluster with the lowest cost.

4. EVALUATION

To evaluate the efficiency and performance of the proposed compressed formats, we implemented the sparse matrix multiplication kernels using the CUDA 5.0 SDK [19] from Nvidia, and measured their run time on different data sets. In our experiments, we are only concerned with the execution time of the kernels and did not take into account the time taken to transfer the data between host memory and the GPU device memory. For our implementation, the parameter h is mapped to the thread block size in a kernel and is currently set to 256, which is also the default size used in [5]. Access to the x input vector is enabled with the help of texture cache.

Table 1: Specifications of the Nvidia GPUs used in evaluation study.

Specifications	Tesla C2070	GTX680	Tesla K20
Compute capability	2.0	3.0	3.5
Cores	448	1536	2496
Mem. BW (GB/s)	144	192.3	208
DP perf. (GFlop/s)	515	129	1170

4.1 Experimental Setup

The SpMV kernels are tested on two GPUs from Nvidia as listed in Table 1. These GPUs were chosen for their different bandwidths and double precision (DP) performance. The first GPU, Tesla C2070, is based on the Fermi architecture, whereas the GeForce GTX680 and Tesla K20 GPUs are based on the newer Kepler architecture. The relevant specifications such as the compute capability[1] and the number of cores of each GPU are also listed in the table. Tesla C2070 contains 448 cores which are organized into 14 Streaming Multiprocessors (SM) of 32 cores each. On the other hand, the cores in GeForce GTX680 and Tesla K20 are organized into Next-generation Streaming Multiprocessors (SMX), with each containing 192 cores. The peak memory bandwidth of GeForce GTX680 at 192 GB/s is greater than that of the Tesla C2070 at 144 GB/s. However, the peak double precision performance of the Tesla C2070 is about 4 times that of the GTX680. Of the three GPUs, Tesla K20 has the highest double precision performance, which is about 2 times that of Tesla C2070, as well as the highest peak memory bandwidth. The measured bandwidths achievable on Tesla C2070, GeForce GTX680, and Tesla K20 are ∼114 GB/s, ∼149 GB/s and 159 GB/s, respectively.

4.2 Experiment Results

4.2.1 SpMV kernel performance analysis

First, we evaluated the performance of the BRO-ELL SpMV kernel to analyze its scalability. In general, the actual SpMV performance depends not only on the amount of memory traffic, but also on the data access patterns of the input vector x. A highly random access to x has low spatial and temporal locality and will cause frequent texture cache misses. To properly evaluate the performance of our kernel as the amount of compression changes, a dense matrix was used in this experiment in order to avoid variations in performance due to cache effects when reading the x vector. To simulate different compression ratios, the number of bits allocated to each index value of a non-zero entry was varied.

Figure 3 shows the performance of the kernel in terms of

[1]This term is used by Nvidia to designate version numbers for different hardware architectures. See Appendix F of the CUDA programming guide [19] for a comparison table of the features of different compute capabilities.

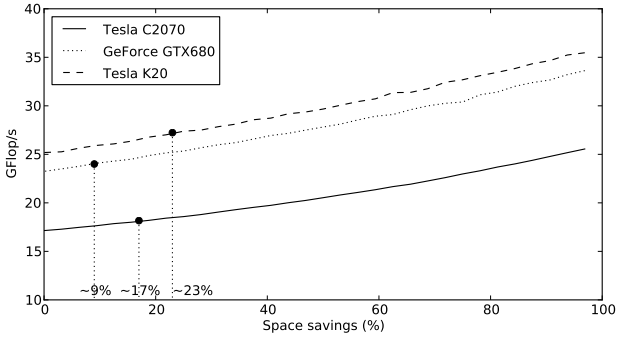

Figure 3: Performance of the BRO-ELL SpMV kernel when the amount of space savings is varied.

billions of floating point operations per second (GFlop/s). The horizontal axis denotes the space savings (η) of the index data due to compression. We define space savings as $1 - C/O$, where C is the compressed size of the index array and O is its original size. Higher space savings results in greater reduction in memory traffic. Compression ratio (κ) is related to space savings by the expression, $\kappa = 1/(1-\eta)$. A higher compression ratio leads to greater space savings.

The graphs in Fig. 3 show that in practice, the performance of the kernel scales linearly with the amount of space savings achieved. A higher performance can be achieved with a larger amount of space savings (i.e. when compression ratio is high). A higher compression ratio means that on average, the number of bits allocated to each index is small, and thus the frequency of memory requests is decreased proportionately. Consequently, the total amount of bytes transferred from global memory is reduced, thereby leading to improved performance. In addition, we can see that the performance from the Tesla K20 GPU is consistently above that of both GeForce GTX680 and Tesla C2070. This is to be expected since sparse matrix-vector multiplication is essentially a memory bandwidth-limited operation, and the pin bandwidth of Tesla K20 is the highest among all the three GPUs.

In the figure, the performance achieved by the ELLPACK format on each of the GPUs is also annotated on each of the lines. Because of the overheads associated with the decompression phase, the figure shows that space savings of at least 17%, 9% and 23% (corresponding to compression ratios of 1.2×, 1.1× and 1.3×) are required to achieve performance gains over ELLPACK.

4.2.2 Benchmarking matrices

A total of thirty real-world matrices listed in Table 2 are used for the subsequent experiments. Many of these matrices were also used in [4, 16, 26] and can be obtained from the University of Florida collection [7]. Note that there was no attempt to select matrices with a particular structure. For each matrix, its dimension, the total number of non-zeros (*nnz*), the average non-zeros per row (μ) and its standard deviation (σ) are given in the table. The matrices are divided into two sets. The first set contains sixteen matrices which can be represented in the BRO-ELL format. The second set contains the remaining matrices which cannot be represented using a BRO-ELL-only format because of large variations in row lengths, but instead can be represented using the BRO-HYB format.

Table 2: Overview of the sparse matrices used in the evaluation study. Total number of non-zeros (*nnz*), average (μ) and standard deviation (σ) of row lengths are shown.

Matrix	Dimensions	*nnz*	μ	σ
Test Set 1				
cage12	130k × 130k	2032536	15.6	4.7
cant	62k × 62k	4007383	64.2	14.1
consph	83k × 83k	6010480	72.1	19.1
e40r5000	17k × 17k	553956	32.1	15.5
epb3	85k × 85k	463625	5.5	0.5
lhr71	70k × 70k	1528092	21.7	26.3
mc2depi	526k × 526k	2100225	4.0	0.1
pdb1HYS	36k × 36k	4344765	119.3	31.9
qcd5_4	49k × 49k	1916928	39.0	0.0
rim	23k × 23k	1014951	45.0	26.6
rma10	47k × 47k	2374001	50.7	27.8
shipsec1	141k × 141k	7813404	55.5	11.1
stomach	213k × 213k	3021648	14.2	5.9
torso3	259k × 259k	4429042	17.1	4.4
venkat01	62k × 62k	1717792	27.5	2.3
xenon2	157k × 157k	3866688	24.6	4.1
Test Set 2				
bcsstk32	45k × 45k	2014701	45.2	15.5
cop20k_A	121k × 121k	2624331	21.7	13.8
ct20stif	52k × 52k	2698463	51.6	17.0
gupta2	62k × 62k	4248286	68.5	356
hvdc2	190k × 190k	1347273	7.1	3.8
mac_econ(_fwd500)	207k × 207k	1273389	6.2	4.4
ohne2	181k × 181k	11063545	61.0	21.1
pwtk	218k × 218k	11634424	53.4	4.7
rail4284	4.3k × 109k	11279748	2633	4209
rajat30	644k × 644k	6175377	9.6	785
scircuit	171k × 171k	958936	5.6	4.4
sme3Da	13k × 13k	874887	70.0	34.9
twotone	121k × 121k	1224224	10.1	15.0
webbase-1M	1M × 1M	3105536	3.1	25.3

Table 3: Space savings (η) achieved in compressing the index data in BRO-ELL matrices.

Matrix	η (%)	Matrix	η (%)
cage12	78.0%	qcd5_4	87.7%
cant	85.9%	rim	92.7%
consph	85.3%	rma10	90.8%
e40r5000	92.5%	shipsec1	92.9%
epb3	83.2%	stomach	70.7%
lhr71	92.1%	torso3	75.9%
mc2depi	50.7%	venkat01	90.2%
pdb1hys	89.2%	xenon2	74.0%

4.2.3 Performance benchmarks

The performance of BRO-ELL SpMV was measured using a variety of real-world matrices from Test Set 1. Note that this set of matrices can be entirely represented using the BRO-ELL-only format. Table 3 shows the amount of space savings (in percentage) that is achieved by the index data of each matrix after compression. Figure 4 shows the results of the experiment comparing the performance of BRO-ELL to ELLPACK for all GPUs. For the sake of comparison, we also included the state-of-the-art ELLPACK-R format.

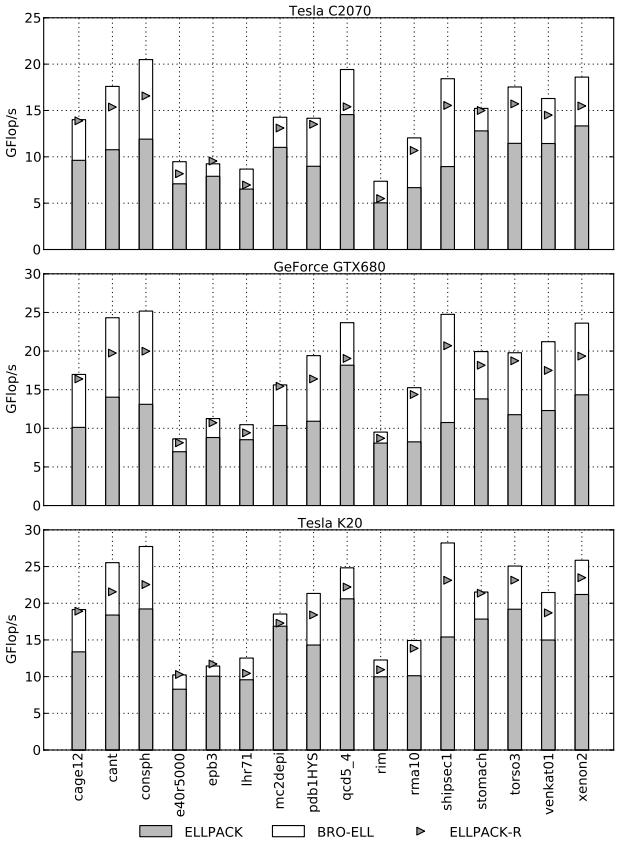

Figure 4: Performance comparison of BRO-ELL to ELLPACK and ELLPACK-R.

In general, we obtained 1.2× to 2.1× speedup over ELLPACK for the collection of sparse matrices tested. On average, a speedup of 1.5× relative to the ELLPACK format was achieved on Tesla C2070, with 1.6× and 1.4× on GTX680 and Tesla K20, respectively. As exemplified in the previous experiment in Section 4.2.1, we can see that in general, real-world matrices whose index data is more compressible result in better performance gains over ELLPACK. For

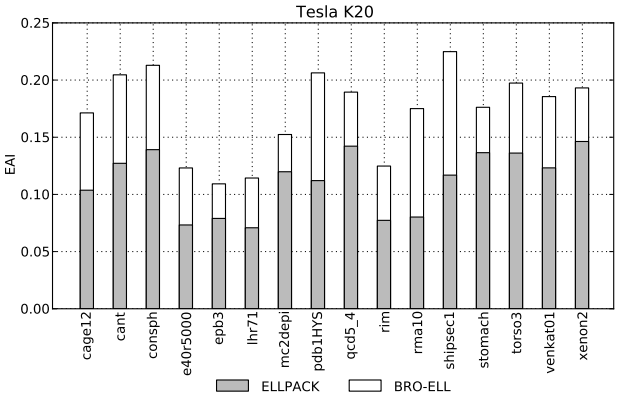

Figure 5: A comparison of the effective arithmetic intensity (EAI) between BRO-ELL and ELLPACK.

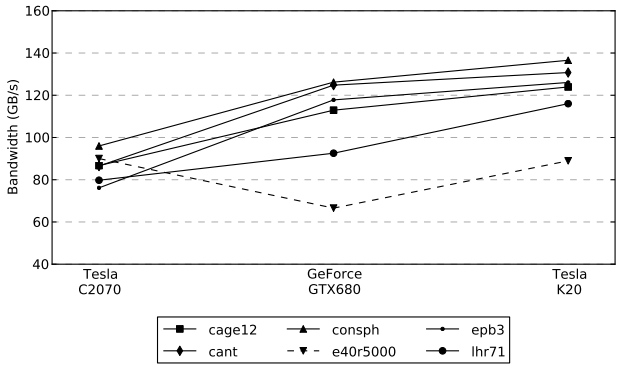

Figure 6: Memory bandwidth utilization by BRO-ELL across different GPUs for the first six matrices.

instance, *consph* and *stomach* have similar uncompressed performance on Tesla C2070. After compression, *consph* achieved a speedup of about 1.7× compared to 1.2× for *stomach*. The former matrix has a higher compression ratio, and therefore greater space savings than the latter matrix. Because the BRO-ELL format is a compressed extension of ELLPACK, we also observed that matrices such as *shipsec1* and *venkat01* which have lower variances in their row lengths generally perform better than those with larger variances such as *lhr71*. Note that overall performance does not only depend on the amount of storage savings alone. For real-world matrices, other factors such as the distribution of the non-zeros in the matrix also play a part by affecting the performance of the texture cache during reading of the x vector. Figure 4 also compares BRO-ELL against the state-of-the-art ELLPACK-R format. On average, BRO-ELL performed 13% faster than ELLPACK-R, and on some matrices such as *consph*, *qcd5_4* and *shipsec1*, speedups of about 1.2× relative to ELLPACK-R were registered.

We define the effective arithmetic intensity (*EAI*) as $\mathcal{F}/\mathcal{B}$, where $\mathcal{F}$ is the number of floating-point operations per second and $\mathcal{B}$ is the kernel memory throughput. In Fig. 5, we display the EAI for both the ELLPACK and BRO-ELL formats on Tesla K20. The comparison shows that BRO-ELL achieved a higher EAI than ELLPACK. This is because BRO-ELL has lower memory bandwidth requirements than ELLPACK due to the compression scheme.

In Fig. 6, we plot and compare the DRAM bandwidth utilization across the different GPUs for the first six matrices. Generally, the BRO-ELL kernel scales with the memory bandwidth of the underlying GPU architecture. However, for the *e40r5000* matrix denoted by dashed lines in the figure, a drop in bandwidth utilization on GTX680 was registered. In addition, the utilization on Tesla K20 was about the same as on Tesla C2070 in spite of its higher theoretical peak bandwidth. The reason for this is because *e40r5000* does not have enough rows to keep the higher number of cores on GTX680 and Tesla K20 sufficiently busy. This is also the reason behind the smaller speedups obtained for both the *e40r5000* and *rim* matrices. Therefore, in order to obtain a good performance for BRO-ELL, a matrix should also have a large number of rows apart from having good compressibility.

In the next experiment, the performance of BRO-COO is evaluated and the results are shown in Fig. 7. All the ma-

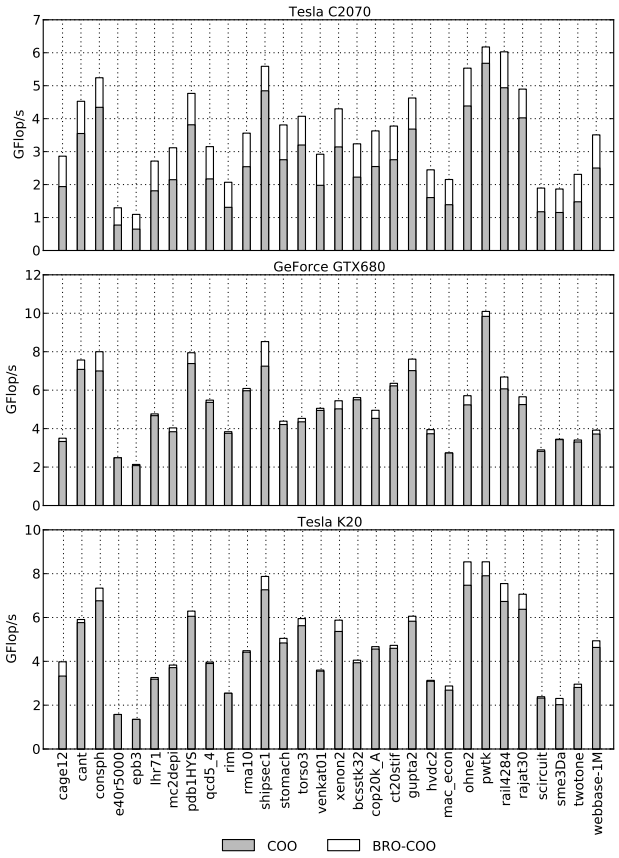

Figure 7: Performance comparison of BRO-COO to COO.

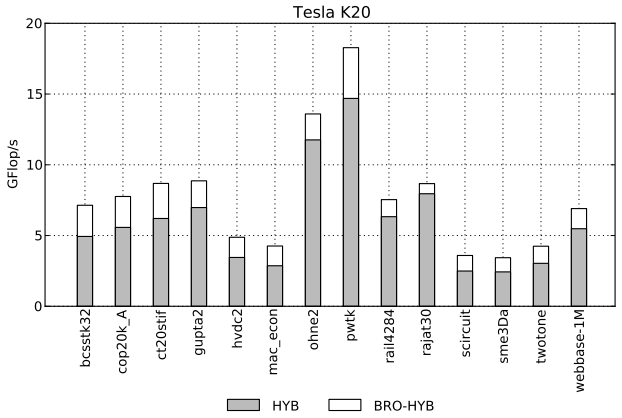

Figure 8: Performance comparison of BRO-HYB to HYB.

trices in Table 2 were used for this evaluation. Since the BRO-COO kernel is computationally more intensive than BRO-ELL owing to the use of the parallel scan primitive, as well an extra kernel invocation for data reduction, we do not expect the average speedup in performance due to compression to be as significant as that achieved by BRO-ELL. As is expected, the results shown in Fig. 7 confirm our hypothesis. We can also see from the figure that speedups obtained on GTX680 and Tesla K20 are considerably less than those obtained on Tesla C2070. For example, *e40r5000*, *epb3* and *rim* did not register any speedup on Tesla K20 unlike on Tesla C2070. This disparity is attributed to architectural differences between the GPUs. The Kepler-based GPUs have higher memory bandwidths as well as different cache hierarchies with lower latencies compared to the Fermi GPU. These advantages resulted in a higher COO baseline performance on the newer architecture, whereas BRO-COO is burdened by the additional need to decompress data on-the-fly.

Next, since matrices in Test Set 2 cannot be efficiently represented using a BRO-ELL-only format, we stored them using BRO-HYB, which partitions each matrix into a BRO-ELL and a BRO-COO component. We then compared the performance of BRO-HYB with the HYB format. Note that for the sake of fair comparison, the matrices are partitioned in the same manner for both HYB and BRO-HYB formats. Table 4 shows the fraction of matrix partitioned into BRO-ELL, and the space savings (η) achieved by BRO-HYB.

Evaluation results for BRO-HYB on Tesla K20 are presented in Fig. 8. The results for Tesla C0270 and GTX680 are similar. Previously, we saw that BRO-ELL performed better than BRO-COO, and in general, the more compressible the matrix, the higher its SpMV performance. Therefore, we can see from Fig. 8 that compressible matrices such as *bcsstk32* and *pwtk* where a large fraction of the matrix is formatted using BRO-ELL achieved higher speedups than other matrices such as *rail4284* or *rajat30* which are less compressible and have lower BRO-ELL fractions. On average, a speedup of 1.6× relative to HYB was achieved on Tesla C2070, with 1.3× and 1.4× on GTX680 and Tesla K20, respectively.

4.2.4 *Effects of BRO-aware reordering*

We evaluated the BRO-aware reordering method (BAR) proposed in Section 3.4 and compared them to reordering methods that are not BRO-aware. Specifically, we compare them with the Reverse Cuthill-Mckee (RCM) and approximate minimum degree (AMD) methods. For this study, the matrices from Test Set 1 were first processed using Algorithm 2 before being compressed using the BRO-ELL for-

Table 4: Partitioning of BRO-HYB matrices into BRO-ELL+BRO-COO and the space savings (η) achieved.

Matrix	% BRO-ELL	η (%)
bcsstk32	96.6%	60.4%
cop20k_A	82.3%	46.7%
ct20stif	90.7%	55.9%
gupta2	50.0%	43.8%
hvdc2	86.9%	45.5%
mac_econ	81.1%	51.6%
ohne2	96.5%	49.5%
pwtk	99.4%	78.7%
rail4284	0.85%	45.2%
rajat30	68.1%	34.5%
scircuit	78.2%	36.6%
sme3Da	83.6%	55.6%
twotone	61.8%	48.8%
webbase-1M	64.2%	13.4%

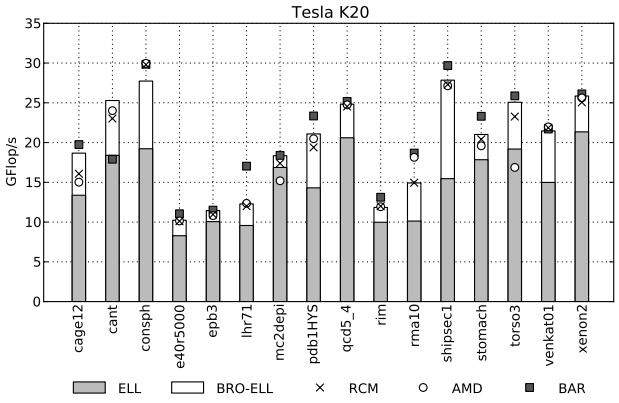

Figure 9: Comparison of BRO-aware reordering to non-BRO-aware RCM and AMD methods.

Table 5: Space savings (η) achieved after reordering with BAR.

Matrix	η (%)	Matrix	η (%)
cage12	81.1%	qcd5_4	88.9%
cant	92.7%	rim	96.0%
consph	91.7%	rma10	94.9%
e40r5000	95.4%	shipsec1	94.8%
epb3	83.2%	stomach	82.3%
lhr71	95.7%	torso3	83.6%
mc2depi	50.7%	venkat01	92.3%
pdb1hys	90.8%	xenon2	87.3%

mat. Figure 9 displays the reordering results and Table 5 lists the space savings achieved after reordering with the BAR method. For the purpose of comparison, we also plot in Fig. 9 the performance achieved by the ELLPACK and BRO-ELL formats before matrix reordering. From the figure, we can see that reordering with BAR resulted in better performance compared to RCM or AMD for majority of the test matrices. On average, with matrix reordering using BAR, we observed additional performance gains of 7% on top of the original BRO-ELL performance. Compare this with slowdowns of about 4% for RCM and AMD which are not BRO-aware. In addition, additional space savings of 4% on average were achieved using BAR compared to 1% reduction in space savings when using RCM or AMD. However, the greedy search algorithm (Algorithm 2) may not always return a good solution. For example in one instance (*cant*), the performance of BAR fared worse than RCM and AMD. This is because a limitation of our model as given in Eqn. (1) is that it does not take into account temporal cache misses that occur when accessing the x vector. Improvements to the current model will be part of our future work. Nevertheless, we find that BRO-aware reordering is important in the context of the BRO-ELL format, and usually performs better than non-BRO-aware reordering.

5. RELATED WORK

Sparse Matrix Computation. A large body of work has been published on SpMV computation on modern processors [12, 17, 20, 24, 27]. However, many of the optimization techniques developed for the CPU cannot be directly applied to the GPU due to differences in their architectures. In particular, implementations for GPUs have to take into account memory coalescing effects and avoid warp divergence to achieve good performances. As such, Baskaran et al. [2] implemented and optimized SpMV for GPUs based on the Compressed Sparse Row (CSR) format. Their targeted platforms were GeForce 8800GTX and GTX280. Bell and Garland [4] developed efficient implementations of SpMV computation for the COO, CSR and ELLPACK formats. They also designed a hybrid ELLPACK-COO format known as HYB that gave the best performance on the GeForce GTX285 for most of the unstructured matrices benchmarked.

Vázquez et al. [23] introduced the ELLPACK-R format which is a variant based on ELLPACK. It tries to improve performance by reducing redundant computation and data access through the use of an additional array that stores the number of non-zeroes in each row of a matrix. The Sliced-ELLPACK format by Monakov et al. [18] reduces redundant computation and storage overhead by reordering rows and partitioning the sparse matrix into slices containing rows of similar lengths. The length of each slice is stored in another array. Their implementation also supports slices with variable heights. Choi et al. [6] experimented with a blocked ELLPACK format and performed model-based tuning to achieve higher performance than ELLPACK. Su and Keutzer [22] developed the *clSpMV* framework that tunes and generates the best *Cocktail* representation, a format that partitions a matrix into submatrices and uses the best representation for each submatrix. Grewe and Lokhmotov [11] developed a code generator that autotunes and generates optimized SpMV kernels for several of the formats discussed above.

Unlike previous works such as ELLPACK-R and Sliced-ELLPACK which mainly focused on reducing redundant computations, our approach differs in that effective bit-representation-optimized compression schemes were designed that result in performance improvement through the reduction of memory traffic.

Compressed Formats. In relation to our work on sparse matrix compression, a few papers have described compressed representations that are designed to be efficient on modern CPUs. Although blocked formats such as [13, 20, 25, 6, 10] can be considered to be compressed in the general sense because only the block index needs to be kept for each dense sub-block of the matrix, they still do not fully exploit the redundancy in the index data, unlike the methods proposed in [3, 16, 26].

In the work by Willcock and Lumsdaine [26], two lossless compression methods for the CSR format were developed to reduce the memory bandwidth required by large sparse matrices. The first method is a hybrid delta and run-length coding scheme that employs a set of six command codes to encode the index data. The second method is an adaptive approach that requires more compression time but performs better. It is based on abstracting and merging groups of interval lists of delta values from the index data. Speedups of up to 30% were achieved using the adaptive method. In another work, Kourtis et al. [16] similarly employed index data compression, but at a coarser-grained level in order to reduce branch mispredictions. Furthermore, they proposed compressing the values data in addition to the index data. Belgin et al. [3] presented a different approach based on iden-

tifying repeating block patterns in a matrix and replacing them with indices to the corresponding bitmask pattern, and achieved average speedups of 40%. These methods target the CPU but are not suitable for the warp-execution model on the GPU due to excessive branching, unlike our proposed compressed formats.

Matrix reordering methods such as the Reverse Cuthill-McKee [9] algorithm have been used to reduce the matrix bandwidth and decrease the number of fill-ins when applying LU or Cholesky factorization. In Pinar and Heath [20], matrix reordering was formulated as a traveling salesman problem to reorder columns to increase the sizes of dense blocks in a row. Monakov et al. [18] used a simple heuristic to order a matrix such that rows with the same number of non-zeros are close to one another. The blocked ELLPACK format by Choi et al. [6] also employs a form of ordering in order to reduce the amount padding required by that format. In contrast with these previous works, in this paper a compression-aware matrix reordering is proposed and formulated as a constrained data clustering problem.

6. CONCLUSION

We have investigated using compression to improve the performance of sparse matrix-vector multiplication on modern GPUs. Although compression has been proposed to improve SpMV performance on the CPU, its use on GPUs has not been demonstrated previously. This is because many of the compression and decompression schemes are serial in nature, and require many branching conditions which are not suitable for GPU architectures. In this work, we have developed effcient bit-representation-optimized (BRO) compression schemes which are suitable for computing the SpMV on GPUs. Experiment results demonstrate that average speedups of 1.5× relative to ELLPACK and HYB can be achieved with our proposed schemes. To further improve the compression ratio, a BRO-aware matrix reordering scheme was formulated as a data clustering problem and a heuristic algorithm was proposed. Results show that on average, BRO-aware reordering performs better than non-BRO-aware schemes. In future, other sources of performance improvement such as assigning multiple threads per row as well as value data compression will be investigated.

Acknowledgments

This work was supported by the Agency for Science, Technology and Research PSF Grant No. 102-101-0028. We are also grateful to the anonymous reviewers for their comments.

7. REFERENCES

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.*, 17(4):886–905, Oct. 1996.

[2] M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs. Technical report, RC24704, IBM T. J. Watson, 2009.

[3] M. Belgin, G. Back, and C. J. Ribbens. Pattern-based sparse matrix representation for memory-efficient SMVM kernels. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 100–109, New York, NY, USA, 2009.

[4] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009.

[5] N. Bell and M. Garland. *Cusp library*, 2012. http://cusp-library.googlecode.com.

[6] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 115–126, New York, NY, USA, 2010.

[7] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011. http://www.cise.ufl.edu/research/sparse/matrices/.

[8] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills. Vectorized sparse matrix multiply for compressed row storage format. In *Proceedings of the 5th international conference on Computational Science - Volume Part I*, ICCS'05, pages 99–106, Berlin, Heidelberg, 2005.

[9] A. George and J. W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall Professional Technical Reference, 1981.

[10] J. Godwin, J. Holewinski, and P. Sadayappan. High-performance sparse matrix-vector multiplication on GPUs for structured grid computations. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 47–56, New York, NY, USA, 2012.

[11] D. Grewe and A. Lokhmotov. Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 12:1–12:8, New York, NY, USA, 2011.

[12] E.-J. Im. *Optimizing the performance of sparse matrix-vector multiplication*. PhD thesis, University of California Berkeley, 2000.

[13] E.-J. Im and K. A. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *Proceedings of the International Conference on Computational Sciences-Part I*, ICCS '01, pages 127–136, London, UK, 2001.

[14] A. K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recogn. Lett.*, 31(8):651–666, June 2010.

[15] D. R. Kincaid, T. C. Oppe, and D. M. Young. *ITPACKV 2D User Guide, CNA-232*, 1989. http://rene.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/.

[16] K. Kourtis, G. Goumas, and N. Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *Proceedings of the 5th Conference on Computing frontiers*, CF '08, pages 87–96, New York, NY, USA, 2008.

[17] J. Mellor-Crummey and J. Garvin. Optimizing sparse matrix-vector product computations using unroll and jam. *Int. J. High Perform. Comput. Appl.*, 18(2):225–236, May 2004.

[18] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *Proceedings of the 5th international conference on High Performance Embedded Architectures and Compilers*, HiPEAC'10, pages 111–125, Berlin, Heidelberg, 2010.

[19] Nvidia Compute Unified Device Architecture (CUDA). http://www.nvidia.com/object/cuda_home_new.html.

[20] A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, Supercomputing '99, New York, NY, USA, 1999.

[21] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[22] B.-Y. Su and K. Keutzer. clSpMV: A cross-platform OpenCL SpMV framework on GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 353–364, New York, NY, USA, 2012.

[23] F. Vázquez, J. J. Fernández, and E. M. Garzón. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurr. Comput. : Pract. Exper.*, 23(8):815–826, June 2011.

[24] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC, J. Physics: Conf. Ser.*, volume 16, pages 521–530, 2005.

[25] R. W. Vuduc and H.-J. Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *Proceedings of the First international conference on High Performance Computing and Communications*, HPCC'05, pages 807–816, Berlin, Heidelberg, 2005.

[26] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 307–316, New York, NY, USA, 2006.

[27] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 38:1–38:12, New York, NY, USA, 2007.